University of Maryland Technical Report UMIACS-TR-2006-36

Application-Level Correctness and its Impact on Fault Tolerance

Xuanhua Li and Donald Yeung
Department of Electrical and Computer Engineering
University of Maryland at College Park
{xli,yeung}@eng.umd.edu

Abstract

Fundamental to any fault tolerance research is the definition of correct program execution. Traditionally, correct program's execution requires architectural state to be numerically perfect. However, in many cases, even if program execution is not 100% numerically correct, it may be completely acceptable if the answers can satisfy user's requirement. Hence, faults which have caused such numerically faulty execution are no longer intolerable.

The extent to which programs are more fault resilient at higher levels of abstraction is application dependent. Programs that produce inexact and/or approximate outputs can be very resilient at the application level. We call such programs soft computations*, and we find they are common in multimedia workloads, as well as artificial intelligence (AI) workloads. Programs that compute exact numerical outputs offer less error resilience at the application level. However, we find* all *programs studied in this paper exhibit some enhanced fault resilience at the application level, including those that are traditionally considered exact computations–e.g., SPECInt CPU2000.*

This report investigates definitions of program correctness that view correctness from the application's standpoint rather than the architecture's standpoint. Under application-level correctness*, a program's execution is deemed correct as long as the result it produces is acceptable to the user. To quantify user satisfaction, we rely on application-level fidelity metrics that capture user-perceived program solution quality. We conduct a detailed fault susceptibility study that measures how much more fault resilient programs are when defining correctness at the application level compared to the architecture level. Our results show for 6 multimedia and AI benchmarks that 45.8% of architecturally incorrect faults are correct at the application level. For 3 SPECInt CPU2000 benchmarks,*

17.6% of architecturally incorrect faults are correct at the application level. Based on our study on algorithmic properties for fault tolerance, we also investigate a lightweight fault recovery mechanism that exploits the relaxed requirements on numerical integrity provided by application-level correctness to reduce checkpoint cost. Our lightweight fault recovery mechanism successfully recovers 66.3% of program crashes in our multimedia and AI workloads, while incurring minimum runtime overhead.

Keywords: Fault Tolerance; Application-Level Correctness; Soft Computing; Fault Injection; Recovery.

1 Introduction

Technology scaling–including feature size, voltage, and clock frequency scaling–has brought tremendous improvements in performance over the past several decades. Unfortunately, these same trends will make computer systems significantly more susceptible to hardware faults in the future, resulting in reduced system reliability. Sources of hardware faults include soft errors [1], wearout [2], and process variations. In anticipation of the reduced reliability that further technology scaling will bring, computer architects have recently focused on several important fault tolerance issues. Areas of focus include characterizing fault susceptibility [3], and developing low-cost fault detection [4, 5, 6, 7, 8] and recovery [9] techniques.

Fundamental to all such reliability research is the definition of correct program execution. In the past, researchers have made very strict assumptions about program correctness. Traditionally, a program's execution is said to be correct only if architectural state is numerically perfect on a cycle-by-cycle basis. A similar (though slightly looser) notion of correctness requires a program's visible architectural state–*i.e.*, its output state–to be numerically perfect. In both cases, correctness requires precise numerical integrity at the architecture level, a fairly strict requirement.

An interesting question is: must we require strict numerical correctness for overall program execution to be correct? In many programs, even if execution is not 100% numerically correct, the program can still *appear* to execute correctly from the user's perspective. Although such numerically faulty executions do not pass the muster of architecture-level correctness, they may be completely acceptable at the user or application level. Hence, whether a fault is intolerable or benign may depend on the *level of abstraction* at which correctness is evaluated. In general, more faults are acceptable at higher levels of abstraction, *i.e.* at the application level, compared to lower levels of

abstraction, *i.e.* at the architecture level.

How much more fault resilient are programs at the application level? The answer to this question is application dependent, and primarily depends on how numerically exact a program's outputs need to be. For instance, programs that process human sensory and perception information are highly fault resilient at the application level. An important example is multimedia workloads. Another example is artificial intelligence workloads (*e.g.*, reasoning, inference, and machine learning), which have become increasingly important recently [10]. These programs belong to a class of computations we call *soft computations* [11, 12].[1] Soft computations compute on approximate data values associated with qualitative results, making them highly fault resilient because errors in numerical results seldomly change the *user's interpretation* of those numerical results. In contrast, programs whose correctness are tied directly to the numerical values they compute may offer little error resilience at the application level. Certain lossless data compression algorithms are examples of such programs. While the degree of error resilience at the application level varies across applications, we find *all* programs studied in this paper exhibit some enhanced fault resilience at the application level, including those that are traditionally considered exact computations–*e.g.*, SPECInt CPU2000.

This paper explores definitions of program correctness that view correctness from the application's standpoint rather than the architecture's standpoint. Under *application-level correctness*, a program's execution is deemed correct as long as the result it produces is acceptable to the user. In other words, correctness depends on the *user's interpretation* of a program's numerical result, not the numerical result itself. To quantify user satisfaction, we rely on application-level fidelity metrics that capture program solution quality as perceived by the user. Because the notion of solution quality is different across applications, our fidelity metrics are necessarily application specific, though applications from the same domain may share common fidelity metrics.

Our goal is to understand how application-level correctness impacts a system's susceptibility to faults, especially transient faults or soft errors. The centerpiece of our work is a detailed fault injection study that quantifies how much more resilient programs are to soft errors at the application level compared to the architecture level. Our study injects 156,205 faults into a detailed architectural simulator, and performs 27,067 separate runs to program completion. For soft computations,

[1]The term "soft computation" is normally used to describe artificial intelligence algorithms. In this paper, we use the term to describe multimedia workloads as well because we find they exhibit similar inexact computing properties as the A.I. algorithms. Note, however, this is not commonly accepted use of the terminology.

we find 45.8% of fault injections that lead to architecturally incorrect execution produce acceptable results under application-level correctness. For SPEC programs, a smaller portion of architecturally incorrect faults, 17.6%, produce acceptable results at the application level. In addition to studying fault susceptibility, we also present a lightweight fault recovery mechanism that exploits the relaxed requirements on numerical integrity provided by application-level correctness to reduce checkpoint cost. Our technique checkpoints some minimum state needed to recover after a crash, but omits from checkpoints those data values for which the user can tolerate numerical imprecision. Although our lightweight fault recovery mechanism is not fail-safe, it successfully recovers 66.3% of program crashes in our multimedia and AI workloads, while incurring extremely low runtime overhead.

The remainder of this paper is organized as follows. Section 2 discusses our definitions of application-level correctness. Then, Section 3 presents our experimental methodology and Section 4 reports on our fault susceptibility study. Next, Section 5 describes our lightweight recovery mechanism. Finally, Section 6 presents related work, and Section 7 concludes the paper.

2 Application-Level Correctness

This section presents our application-level correctness definitions. We begin by discussing soft program outputs, an important property for application-level correctness (Section 2.1). Then, we present fidelity metrics that quantify application-level correctness for the benchmarks studied in this paper (Section 2.2). Finally, we discuss limitations of our approach (Section 2.3).

2.1 Soft Program Outputs

Programs can exhibit enhanced error resilience at the application level compared to the architecture level for many reasons. However, the likelihood of this happening increases when a program permits *multiple valid outputs*. In this paper, we say such programs have "soft outputs." Soft outputs commonly occur in programs computing results that are interpreted qualitatively by the user. Different numerical results can lead to the same or similar qualitative interpretation. Hence, multiple numerical outputs may be acceptable to the user. Another source of soft program outputs is heuristic-based algorithms. Many programs solve complex problems for which optimal solutions are unachievable. Instead of the optimal, they try to find the best solution possible given available computational resources. In practice, many solutions are "good enough." So, once again, multiple numerical outputs are acceptable to the user.

Benchmark	Numerical Output	Qualitative Output	Fidelity Metric
Multimedia			
G.721-D	Decompressed audio datafile	Perceived audio	Segmental Signal-to-Noise Ratio (SNRseg)
JPEG-D	Decompressed image datafile	Perceived image	Peak Signal-to-Noise Ratio (PSNR)
MPEG-D	Decompressed video datafile	Perceived video	Peak Signal-to-Noise Ratio (PSNR)
Artificial Intelligence			
LBP	Markov network belief values	Web Page Class Types	% Classification Change
SVM-L	Support Vector Model	Test Data Class Types	% Classification Change
GA	Thread Schedule	-	% Schedule Length Change
SPECInt CPU2000			
164.gzip	Compressed file	-	Compression Ratio
256.bzip2	Compressed file	-	Compression Ratio
175.vpr	Cell placement	-	Consistency Check

Table 1. Numerical and qualitative outputs computed by our benchmarks. The last column lists the fidelity metrics used to quantify solution quality.

To illustrate the soft output property, Table 1 lists 9 benchmarks used in our study–three from the multimedia domain, three from the artificial intelligence (AI) domain, and three from SPECInt CPU2000. The multimedia workloads, G.721-D, JPEG-D, and MPEG-D are taken from the Mediabench suite [13], and perform audio, image, and video decompression, respectively. All three decompression algorithms are lossy. The AI workloads are from various sources. LBP performs Pearl's Loopy Belief Propagation [14], a well-known message-passing algorithm for approximate inference on large Markov networks. Our LBP implementation solves Taskar's Relational Markov Network applied to a web-page classification problem [15]. SVM-L is the learning portion of a Support Vector Machine algorithm, called SVMlight [16]. SVM-L learns the parameters for a support vector (SV) model on a training dataset. GA is a genetic algorithm applied to multiprocessor thread scheduling [17]. Given a thread dependence graph, GA searches for a thread schedule that minimizes execution time. Finally, the SPECInt CPU2000 workloads are 164.gzip and 256.bzip2, two lossless data compression algorithms, and 175.vpr, a place-and-route program. (The data inputs we use for vpr only perform placement–see Table 3 in Section 3).

The second column of Table 1 reports the numerical outputs computed by each benchmark. Many (in fact, as we will show, *all*) of these numerical outputs are soft, so multiple valid outputs exist. In most cases, the soft outputs are due to the qualitative nature of the program results. When appropriate, we indicate this in the third column, labeled "Qualitative Output." Many of

our benchmarks also achieve soft outputs because they are heuristic-based; some examples of this are discussed below. As we will see in Section 4, the soft computations (multimedia and AI) exhibit soft program outputs to a greater extent than the more traditionally exact computations (SPEC).

For the three multimedia programs, the numerical outputs are the decompressed datafiles, either in audio, image, or video format. Once decompressed, these datafiles can be played back to the user; hence, the qualitative output of these programs is the perceived playback, either aural or visual, of the numerical outputs. Because the user's playback experience is qualitative in nature, it is possible for different numerical outputs to be acceptable (*i.e.*, valid) to the user.

Like the multimedia workloads, the AI workloads also exhibit soft program outputs. In LBP, nodes in the Markov network contain probability distribution functions (PDFs) over the possible class types inferred for web pages. Each PDF encodes how strongly we "believe" a particular web page belongs to each class type. The numerical output for LBP, hence, is the collective belief values across the entire Markov network. In SVM-L, the numerical output is the SV model parameters learned from the training dataset, as described earlier. Both LBP and SVM-L's numerical outputs are soft because they are used to derive classification answers, the qualitative output for these programs. LBP selects a class type for each web page by choosing the most likely class indicated by the corresponding PDF. For SVM-L, extracting class types is more involved because SVM-L itself doesn't perform classification. To obtain the class types we want, we run a separate SVM classifier (not listed in Table 1) that uses the SV model computed by SVM-L to perform classification on a test dataset. Computing the classification answers in both LBP and SVM-L is an extremely inexact process. Multiple numerical outputs (belief values for LBP and SV model parameters for SVM-L) can lead to the same (and hence, valid) classification answer. In GA, the numerical output is the thread schedule it computes. GA's numerical output does not have a qualitative interpretation; however, users can still accept multiple numerical outputs because GA is a heuristic algorithm that produces a "good enough answer." Although it is infeasible to find the optimal thread schedule, there are many thread schedules that are adequate. Any one of these good enough answers represent a valid numerical output from the user's perspective.

Somewhat surprisingly, the three SPEC program outputs are also soft, though we do not call the SPEC benchmarks soft computations. As indicated in Table 1, none of the SPEC outputs have qualitative interpretations; nonetheless, multiple numerical outputs are valid. For the data compression algorithms, there is flexibility in how datafiles are compressed even though the com-

pression algorithms themselves are exact. We will discuss the reasons for this in Section 4. The vpr benchmark tries to find a cell block placement for a chip design. Like GA, vpr's algorithm is heuristic-based since finding an optimal placement (one that minimizes interconnect distance) is intractable. Hence, multiple cell block placements are valid.

2.2 Solution Quality

Because the benchmarks in Table 1 permit multiple valid numerical outputs, their correctness is not simply "black or white;" hence architecture-level correctness (where all architectural values are either correct or wrong) is clearly too strict. An appropriate correctness definition should accommodate all valid numerical outputs. At the same time, it is important to recognize not all valid outputs are of equal value; instead, there are varying degrees of solution quality across our programs' outputs.

We use application-specific fidelity metrics to capture the quality of a program's output as perceived by the user. Our fidelity metrics quantify how different a particular output is relative to a baseline output. (For our experiments in Sections 4 and 5, we define the baseline output to be the result obtained from a fault-free execution of a given benchmark). Outputs that are very similar to the baseline have high fidelity, whereas outputs that are very dissimilar have low fidelity. Whenever possible, we compute fidelity in terms of a benchmark's qualitative outputs instead of its numerical outputs. This enables us to capture fidelity of the user's qualitative experience, an important correctness consideration for many of our benchmarks.

The last column in Table 1 lists the fidelity metrics we use for our 9 benchmarks. For the multimedia workloads, we use signal-to-noise ratio (SNR), a well-accepted fidelity metric in signal processing. Specifically, we use segmental SNR for G.271-D, and peak SNR for JPEG-D and MPEG-D. For LBP and SVM-L, we use the percentage change in classification answers, and for GA, we use the percentage change in thread schedule length (*i.e.*, execution time). For the two data compression algorithms, we use the compression ratio. Lastly, vpr's fidelity metric is a consistency check provided by the code itself. This consistency check first determines whether a given cell block placement is valid (*i.e.*, doesn't violate any design rules), and second computes a cost metric that reflects the degree to which interconnect distance is minimized.

Given the fidelity metrics in Table 1, users can set the minimum fidelity necessary for a program output to be "acceptable," thus allowing users to customize the strictness of output acceptability to

their correctness needs. Notice, users can accept more program outputs as "correct" by sacrificing fidelity. As we will see in Section 4, this provides users with the unique opportunity to tradeoff solution quality for fault tolerance.

2.3 Limitations

A limitation of application-level correctness is it only considers program outputs visible to the user. It does not account for other correctness issues unrelated to visible program outputs. For example, we do not consider real-time issues. Certain errors may not degrade solution quality appreciably, but they may alter *when* solutions become available. This is unacceptable for the correctness of real-time systems. In addition, we do not consider system-level issues. Errors that do not defeat individual benchmarks may still propagate to other programs in a multiprogrammed environment, causing them to crash or execute incorrectly. Lastly, it may still be necessary to provide architecture-level correctness in cases where architecture state is exposed to the user. For example, this is necessary during program debugging. In all these cases, application-level correctness is not strict enough and does not provide the desired correctness requirements.

3 Experimental Methodology

Having presented our definitions of application-level correctness, we now investigate their impact on fault susceptibility. Our goal is to quantify how much more fault resilient programs are under application-level correctness compared to architecture-level correctness. This section discusses the experimental methodology used in our fault susceptibility study. Later, in Section 4, we will present the study's results.

To analyze fault susceptibility, we conduct fault injection experiments [18, 8, 19] to observe the effects of faults on a CPU under different definitions of correctness. Each of our fault injection experiments injects a single bit flip into the execution of one of our benchmarks–*i.e.*, we assume a single event upset, or SEU, fault model. Our approach closely follows the methodology introduced by Reis *et. al.* [8] which uses an efficient two-phase simulation technique. We initially inject faults into a detailed architectural simulator that models a modern out-of-order superscalar pipeline. After each fault is injected, we simulate the microarchitecture until the effect of the fault completely manifests itself in architectural state. Then, we checkpoint the simulator's architectural state, and resume simulation from the checkpoint using a simple functional simulator. We try to run the

Processor Parameters	
Bandwidth	8-Fetch, 8-Issue, 8-Commit
Queue size	64-IFQ, 40-Int IQ, 30-FP IQ, 128-LSQ
Rename register / ROB	128-Int, 128-FP / 256 entry
Functional unit	8-Int Add, 4-Int Mul/Div, 4-Mem Port, 4-FP Add, 2-FP Mul/Div
Branch Predictor Parameters	
Branch Predictor	Hybrid 8192-entry gshare/2048-entry Bimodal
Meta Table / BTB / RAS Size	8192 / 2048 4-way / 64
Memory Parameters	
IL1 config	64kbyte, 64byte block size, 2 way, 1 cycle latency
DL1 config	64kbyte, 64byte block size, 2 way, 1 cycle latency
UL2 config	1Mkbyte, 64byte block size, 4 way, 20 cycle latency
Mem config	300 cycle first chunk, 6 cycle inter chunk

Table 2. Parameter settings for the detailed architectural model into which we inject faults.

benchmark to completion under the functional CPU model, and assuming the benchmark doesn't crash, we evaluate the program's outputs under both architecture- and application-level correctness.

In the detailed simulation phase, we use a modified version of the out-of-order processor model from Simplescalar 3.0 for the PISA instruction set [20], configured with the simulator settings listed in Table 2. Compared to the original, our modified simulator models rename registers and issue queues separately from the Reservation Update Unit (RUU). Using this processor model, we inject faults into three hardware structures: the physical register file, the fetch queue, and the issue queue (IQ).[2] We assume faults injected into a physical register will appear in architectural state unless the register is idle or belongs to a mispeculated instruction. For the fetch queue, we allow faults to corrupt instruction bits, including opcodes, register addresses, and immediate specifiers. These faults manifest in architectural state as long as the injected instruction is not mispeculated. Lastly, for the IQ, we model 6 fields per entry: instruction opcode, 3 register tags (2 source and 1 destination), an immediate specifier, and a PC value. Like the fetch queue, faults in the IQ appear in architectural state for instructions that are not mispeculated. Corruptions to the IQ opcode and immediate fields behave similarly to corresponding corruptions in the fetch queue. Corruptions to the register tags alter instruction dependences, and corruptions to the PC value affect branch target addresses.

When simulating in detailed mode, two issues affect the collection of checkpoints for subsequent

[2]For both the physical register file and issue queue, our simulator models separate integer and floating point versions of the structures. However, when injecting faults, we distribute the faults uniformly across both versions as if they formed a unified structure.

Benchmark	Input	Exec Time	Inter-Arrival	Injects	Regfile	Fetch	Issue
G.721-D	clinton.pcm	77643471	7000.0	10467	483	581	1183
JPEG-D	lena.ppm	44520776	7000.0	5950	542	4341	1483
MPEG-D	mei16v2.m2v	40457756	7000.0	5423	713	434	803
LBP	WebKB [15]	2175526139	1000000.0	2198	1317	946	589
SVM-L	LIBSVM(a1a) [21]	53981768	7000.0	7225	1138	2327	1564
GA	r16-0.1.in [17]	127490411	15000.0	8491	479	626	1352
164.gzip	test	93396309	15000.0	6693	467	829	861
256.bzip2	train	732651712	250000.0	2941	264	1559	722
175.vpr	test	800450837	250000.0	3177	968	166	330

Table 3. Detailed fault injection statistics for benchmarks used in our study. "Exec Time" reports the execution time in cycles for each benchmark. "Inter-Arrival" reports the average time between fault injections. "Injects" reports the total number of faults injected into the physical register file. The last 3 columns report the number of functional simulation runs for the physical register file, fetch queue, and issue queue, respectively.

functional simulation. First, not all fault injections require functional simulation to program completion. Some faults are masked by the microarchitecture, and do not propagate to architectural state. Other faults cause the microarchitecture simulation to incur an exception or lockup. (We rely on a watchdog timer to detect lockups). In these cases, we simply record the outcome, and skip the functional simulation phase. Second, faults in the out-of-order portion of the processor pipeline (*i.e.*, the physical register file and issue queue) can manifest in architectural state in an imprecise manner. For example, a corrupted register value may propagate to some instructions (those that haven't issued yet) but not to others (those that have already issued). Our detailed simulator captures these out-of-order effects, and records them in the checkpoint. Then, when simulating the initial instructions in functional mode (*i.e.*, those that were in-flight at the time of the fault), we propagate the injected fault to exactly the same instructions that were affected during out-of-order simulation.

Table 3 presents detailed fault injection information for each of our benchmarks described in Section 2. The column labeled "Input" specifies the input dataset used for each benchmark, and the column labeled "Exec Time" reports each benchmark's measured execution time in cycles on our detailed out-of-order simulator. We inject faults only after program initialization, so "Exec Time" does not include the benchmarks' initialization phase. After program initialization, we run each benchmark to completion in our detailed simulator, performing all fault injections and checkpoints for a single hardware structure in the same run. We perform 3 such injection runs on each benchmark to inject faults into the 3 hardware structures (physical register file, fetch queue,

and issue queue). In each run, faults are randomly injected into a single hardware structure one after another using a uniformly distributed inter-fault arrival time.

It is crucial to limit the total number of fault injections since each fault potentially requires functional simulation to program completion. Our methodology limits the number of injected faults in two ways. First, we choose program inputs that do not result in exceedingly long execution times. Second, we set the inter-fault arrival time based on each benchmark's execution time. We use larger arrival times for longer-running benchmarks, thus reducing the number of injected faults for benchmarks with longer execution times. The column labeled "Inter-Arrival" in Table 3 reports the inter-fault arrival time used for each benchmark, while the column labeled "Injects" reports the total number of injected faults for the physical register file. (The number of injected faults for the other two hardware structures is very similar since they use the same inter-fault arrival time. More specifically, the total number of injected faults is 52,555 for physical register file, 52,229 for fetch buffer, and 51,421 for issue queue). Across all 3 hardware structures, our fault injection campaign performs 156,205 fault injections.

4 Fault Susceptibility

Our first result is only a portion of fault injections manifest themselves in architectural state because many faults are masked at the microarchitecture level. Microarchitecture-level masking arises due to faults that attack idle hardware resources, or hardware resources occupied by mispeculated instructions. The last three columns in Table 3 report the number of faults injected into the physical register file, fetch queue, and issue queue, respectively, that become architecturally visible. The percentage of effective faults can be computed as a fraction of the total injected faults (*i.e.*, the column labeled "Injects"). As a result, we find that the degree of masking at the microarchitecture level varies considerably across different benchmarks and hardware structures. But on average, only 17.3% of injected faults (27,067 out of 156,205) become architecturally visible, with the fetch queue exhibiting the most fault sensitivity (22.6% visible) and the register file and issue queue exhibiting less sensitivity (12.1% and 17.3% visible, respectively). Faults that are masked by the microarchitecture produce correct program outputs under both architecture- and application-level correctness.

Next, we examine the architecturally visible faults in more detail. Figure 1 breaks down the outcome of all architecturally visible fault injections when they are simulated to program comple-

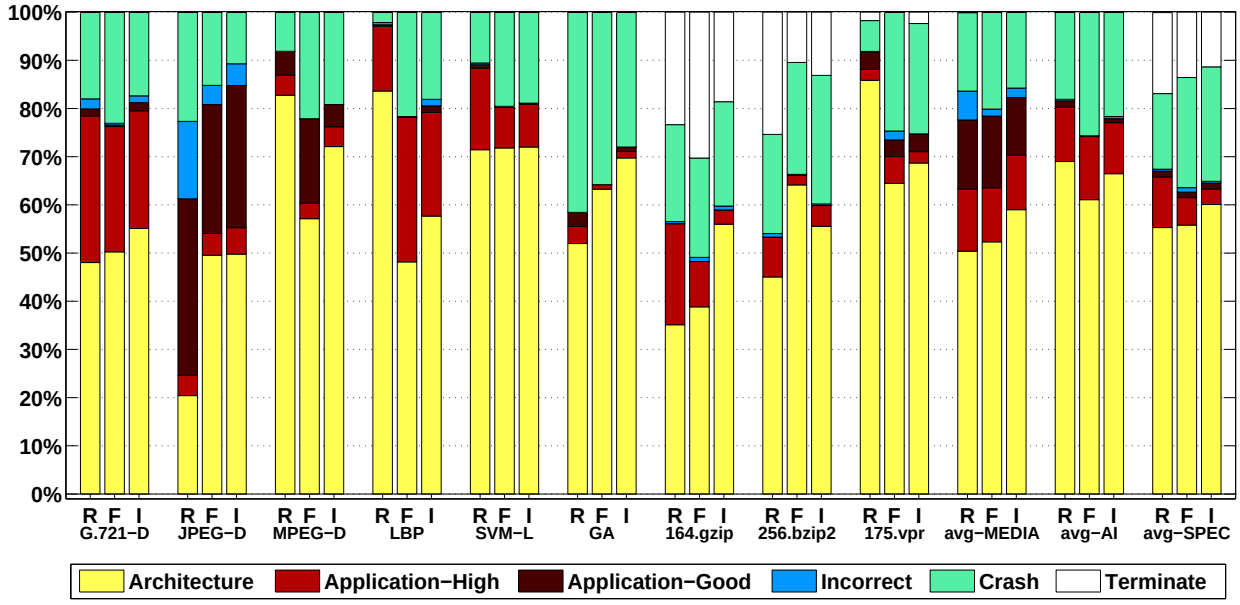

Figure 1. Breakdown of program outcomes for architecturally visible fault injections.

tion. For each benchmark, we report the fault injections into the physical register file, fetch queue, and issue queue separately in a group of 3 bars labeled "R," "F," and "I," respectively. Each bar contains 6 categories. The first category, labeled "Architecture," indicates the program outputs that pass architecture-level correctness (these outputs are also correct at the application level). The next two categories, labeled "Application-High" and "Application-Good," indicate the additional program outputs that are acceptable under application-level correctness only. The category labeled "Incorrect" indicate outcomes that are incorrect under both architecture- and application-level correctness. Finally, the last two categories indicate experiments that fail to complete during functional simulation due to an exception or a hardware lockup (labeled "Crash") or early program exit with an error (labeled "Terminate"). The last 3 groups of bars in Figure 1 report the average breakdowns for the multimedia, AI, and SPEC benchmarks, respectively.

Recall from Section 2.2 that application-level correctness reflects a user's willingness to accept degraded solution quality (as measured by fidelity metrics) in return for error resilience; hence, application-level correctness is user dependent. In our experiments, the "Application-High" category reflects program outputs that maintain very high fidelity with the original (no noticeable solution quality degradation), while the "Application-Good" category reflects outputs with good fidelity with the original (only slightly degraded solution quality). Quantitatively, we distinguish

these categories in the following manner. For all of the fidelity metrics in Table 1 except PSNR, the "Application-High" and "Application-Good" outputs are within 1% and 5%, respectively, of the program outputs obtained via fault-free execution. For the PSNR metric, the "Application-High" and "Application-Good" outputs are greater than 90dB and between 50dB and 90dB, respectively, when compared to outputs from fault-free execution.

Looking at Figure 1, we see a large portion of architecturally visible faults lead to correct program outputs under architecture-level correctness (*i.e.*, the "Architecture" components). The last 3 groups of bars in Figure 1 show architecture-level correctness is achieved in 50.4% to 60.0% of program outputs on average across the 3 hardware structures for the multimedia and SPEC benchmarks, and in 61.0% to 68.8% on average for the AI benchmarks. Similar to microarchitecture-level masking, many fault injections attack architectural state unnecessary for maintaining numerical integrity in our computations, and hence, become architecturally masked. In our benchmarks, the primary source of architecture-level masking is logical and inequality instructions. These instructions seldomly change their output despite data corruptions to their input operands; thus, they are highly resilient to our fault injections. Other (less significant) sources of architecture-level masking include dynamically dead code, NOP instructions, and Y-branches [22]. Both microarchitecture- and architecture-level masking have been previously observed by other fault susceptibility studies [3, 9, 22].

The remaining fault injections that are not masked at the microarchitecture or architecture levels do not produce numerically correct program outputs. These fault outcomes have traditionally been considered *incorrect* under architecture-level correctness. Across all benchmarks and all hardware structures, 41.2% of architecturally visible fault injections on average are architecturally incorrect. However, we find a significant portion of architecturally incorrect outcomes produce high-quality solutions. This is particularly true for the multimedia and AI benchmarks, our soft computations. As the first group of average bars in Figure 1 show, 55.0%, 54.8%, and 56.8% of architecturally incorrect faults for multimedia benchmarks occurring in the physical register file, fetch queue, and issue queue, respectively, produce program outputs with either high or good fidelity (*i.e.*, the "Application-High" or "Application-Good" components). As the second group of average bars show, 40.4%, 33.8%, and 34.0% of architecturally incorrect faults for AI benchmarks occurring in the same three hardware structures, respectively, produce high or good fidelity program outputs as well. While these program outputs are incorrect numerically, they are completely acceptable from

the user's standpoint–*i.e.*, they are correct at the application level. Overall, 45.8% of architecturally incorrect faults in our soft computations achieve application-level correctness.

Looking at the per-application breakdowns for soft computations in Figure 1, we see in many cases one half or more of the architecturally incorrect faults are correct under application-level correctness. This result is fairly consistent across all benchmarks and hardware structures, demonstrating the ability of soft program outputs to mask additional faults. One notable exception is GA where very few additional faults are correct at the application level. As described in Section 2.1, the heuristic search for a good thread schedule performed in GA is quite fault resilient. However, upon closer examination, we found GA spends most of its time evaluating an objective function that reflects the cost of a given thread schedule. Unfortunately, this objective function is not a soft computation, thus reducing the benefits of application-level correctness.

In addition to soft computations, we find our SPEC benchmarks exhibit enhanced fault resilience at the application level as well. As the last group of bars in Figure 1 shows, 26.2%, 15.5%, and 11.1% of architecturally incorrect faults for the SPEC benchmarks occurring in the physical register file, fetch queue, and issue queue, respectively, produce program outputs with either high or good fidelity. As mentioned in Section 2.1, gzip and bzip2's program outputs are soft due to flexibility in how datafiles can be compressed. Upon closer examination, we found certain faults cause these compression algorithms to emit different output tokens compared to a fault-free execution. While these output tokens do not achieve as high a compression ratio, they still correctly encode their corresponding input tokens. Hence, a numerically different (slightly larger) compressed file is created, but the exact original file can still be recovered via decompression. In vpr, as already discussed in Section 2.1, the source of soft program outputs is multiple valid cell block placements. Some of our fault injections cause vpr to produce these different cell block placements. Overall, Figure 1 shows the SPEC benchmarks offer less additional fault resilience at the application level compared to soft computations. However, we believe the fact that application-level correctness provides any additional fault resilience in SPEC is a positive result given these benchmarks are traditionally considered to be exact computations.

5 Fault Recovery

Section 4 demonstrates many architecturally incorrect faults are acceptable when evaluated at the application level. However, even after considering application-level correctness, a large number

of faults still lead to incorrect program outcomes–*i.e.*, the "Incorrect," "Crash," and "Terminate" components in Figure 1. Of these, by far the most significant contributor is the "Crash" component. In all but three bars (the "R" and "F" bars for gzip, and the "R" bar for bzip2), the "Crash" component dominates. Across all benchmarks and all hardware structures, crashes account for 80.8% of faults on average that are incorrect at both the architecture and application levels. Techniques that can address crashes will have a large impact on fault tolerance as fault rates increase in the future.

Addressing crashes requires detecting the corresponding faults, and recovering from them. Since crashes consist of exceptions and program lockups, detection is straight-forward: exceptions are intercepted by the operating system while lockups can be flagged by a CPU watchdog timer. No significant hardware support nor runtime overhead need be incurred for detection. Recovery, on the other hand, can be more costly. Normally, recovery is performed via checkpoints. However, checkpoints incur runtime overhead for copying, either at pre-determined checkpoint locations, or upon first writes (*e.g.*, copy-on-write schemes).

5.1 Lightweight Recovery Mechanism

Soft program outputs, which are responsible for the fault resilience improvements demonstrated in Section 4, can also help reduce the cost of checkpoint recovery. While recovering all modified data is necessary for architecture-level correctness, it is overly conservative for application-level correctness because program outputs no longer need to be numerically perfect. To achieve application-level correctness after a crash, we only need to checkpoint enough state to restart program execution; state that only affects soft program outputs need not be checkpointed, thus reducing both checkpoint size and runtime overhead.

The key question is how do we identify the state that requires checkpointing to achieve application-level correctness? We have examined several program crashes, and found in most cases that program restart can occur simply with a valid program counter (PC) plus the correct stack state at the associated program control point. Hence, we developed a lightweight recovery mechanism that periodically checkpoints the PC, architected register file, and program stack. Upon a crash, we restart the program at the nearest checkpoint, rolling back its PC, register file, and stack only–we do not touch the program text, static data, or heap during rollback. To determine when checkpoints are taken, we identify the main controlling loops in our benchmarks, and instrument checkpointing

Benchmark	# Check	Interval	Size
G.721-D	261	1003622	826 (0.0133)
JPEG-D	59	503137	3034 (0.0033)
MPEG-D	45	2901005	388 (0.0009)
LBP	50	47197236	960 (0.0001)
SVM-L	430	404591	2208 (0.0019)
GA	300	1108510	15642 (0.0003)
164.gzip	252	964376	1368 (0.0004)
256.bzip2	1532	2015753	3722 (0.0004)
175.vpr	2995	505182	3980 (0.0174)

Table 4. Checkpoint statistics. The last 3 columns report the total number of checkpoints, average checkpoint interval size (in instructions), and average checkpoint size (in bytes), respectively.

at the top of each loop iteration. (In our benchmarks, these are the outer loops associated with major program phases; they are not the inner loops).

Notice our lightweight recovery mechanism cannot successfully recover all crashes because it does not guarantee all the state necessary for program restart gets checkpointed. A fail-safe version of our mechanism would need to precisely identify the state associated with soft program outputs, and only omit these data from checkpoints. While our current lightweight recovery mechanism is not fail-safe, as the next section will show, it is very inexpensive, and enables recovery from a significant number of crashes in many cases.

5.2 Recovery Results

We evaluate our lightweight recovery mechanism using the functional simulator from our two-phase simulation methodology (see Section 3). First, we run checkpoint-instrumented versions of our benchmarks on the functional simulator once to acquire all the checkpoints. Table 4 reports statistics from these checkpoint runs. The columns labeled "# Check," "Interval," and "Size" report the total number of checkpoints, the average number of instructions between checkpoints (excluding instrumentation code), and the average checkpoint size, respectively. In parenthesis, we also report the average checkpoint size as a fraction of the total program size. Because we only checkpoint the PC, register file, and stack, our checkpoints are extremely lightweight. On average, our checkpoints are roughly 3 Kbytes in size, with consecutive checkpoints separated by 400,000 instructions or more. Since we acquire our checkpoints on the functional simulator, we have not measured the actual runtime cost of our checkpoints; however, we estimate a 1% runtime overhead at worst.

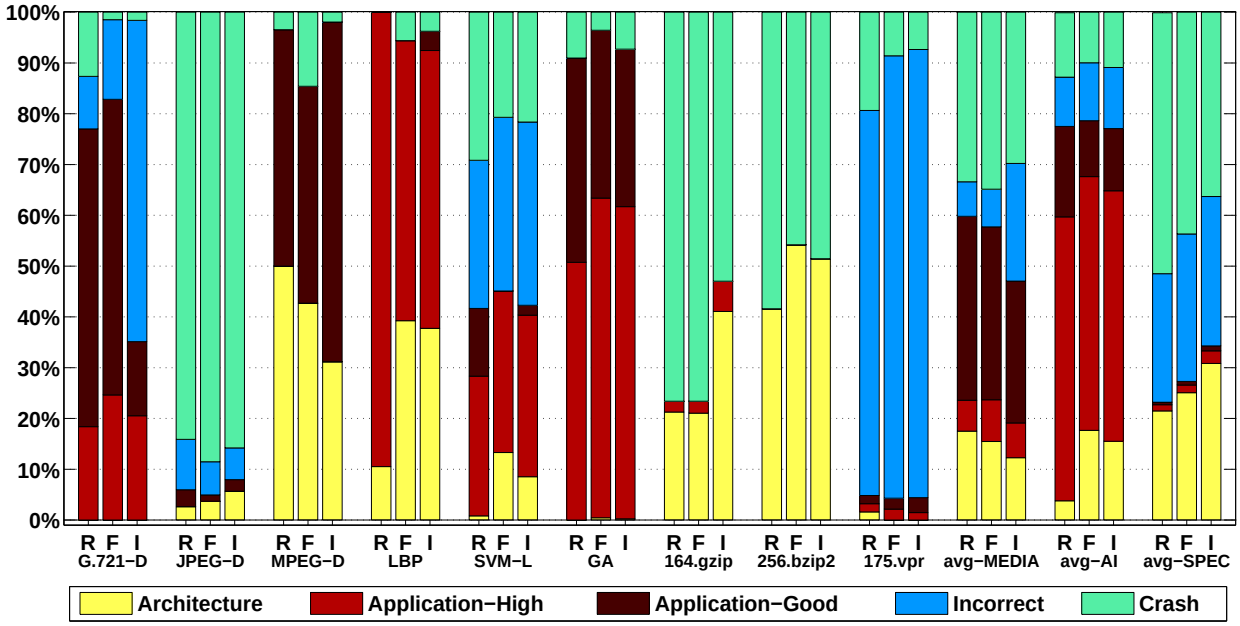

Figure 2. Breakdown of program outcomes for lightweight recovery of crashes.

After acquiring all the checkpoints, we perform recovery experiments. For every crash outcome in Figure 1, we rollback to the nearest checkpoint, as described in Section 5.1, and restart execution in our functional simulator. Then, we try to run the benchmark to completion, and assuming the benchmark doesn't crash again, we evaluate the program's outputs under both architecture- and application-level correctness, just as we did in Section 4. Figure 2 breaks down the outcome of our recovery experiments. For each benchmark, we report the recovery outcome for crashes from the physical register file, fetch queue, and issue queue fault injections separately in a group of 3 bars labeled "R," "F," and "I," respectively. Each bar is broken down into the same categories as Figure 1 minus the "Terminate" category (none of our recovery experiments end in early program exit). The last 3 groups of bars in Figure 2 report the average breakdowns for the multimedia, AI, and SPEC benchmarks, respectively.

Looking at Figure 2, we see a number of recoveries lead to correct program outputs even under architecture-level correctness (*i.e.*, the "Architecture" components). The 3 groups of average bars in Figure 2 show architecture-level correctness is achieved in 3.8% to 17.7% of recoveries on average for the multimedia and AI benchmarks, and in 21.5% to 30.8% of recoveries on average for the SPEC benchmarks. In these cases, there are no corruptions to uncheckpointed state between the rollback checkpoint and the crash; hence, lightweight recovery allows program completion with

numerically perfect outputs. This result shows lightweight recovery can address a modest number of crashes even when strict numerical correctness is necessary.

However, Figure 2 also shows that under application-level correctness, a significant number of additional crashes can be recovered (*i.e.*, the "Application-High" and "Application-Good" components), especially for soft computations. The first 2 groups of average bars in Figure 2 show application-level correctness permits an additional 34.8% to 73.8% of recoveries on average to be correct for the multimedia and AI benchmarks. Averaged across all hardware structures, an additional 52.6% of recoveries are correct under application-level correctness for the soft computations. G.721-D, LBP, and GA respond particularly well to lightweight recovery, with as many as 90% of crash recoveries achieving application-level correctness. In combination with numerically correct recoveries, these additional application-level correct recoveries allow 66.3% of all crashes on average to complete with acceptable results for soft computations. Furthermore, when combined with the results from Figure 1, our lightweight recovery mechanism allows 92.4% of all architecturally visible fault injections to complete with correct outputs at either the architecture or application level for soft computations.

While lightweight recovery performs well for soft computations, the results are not as good for the SPEC benchmarks. Looking at the last group of bars in Figure 2, we see application-level correctness provides at most an additional 2.5% correct outputs on top of the numerically correct recoveries. This lower rate of recovery is due to the fewer soft program outputs allowed by SPEC programs compared to soft computations. Overall, however, Figure 2 illustrates the ability of soft program outputs to enable a large number of user-acceptable recoveries even when checkpointing a small amount of program state. The benefit is greatest for applications that permit a large number of valid program outputs (*i.e.*, soft computations).

6 Related Work

Our work is related to the significant body of prior research on characterizing soft error susceptibility. Several researchers have injected faults into detailed CPU models to investigate soft error effects. Saggese *et al* [23] inject faults into a DLX-like embedded processor; Wang et al. [19] inject faults into a CPU similar to the Alpha 21264 or AMD Athlon; Kim and Somani [18] inject faults into Sun's picoJava-II; and Czeck and Siewiorek [24] inject faults into an IBM RT PC processor. All of these fault susceptibility studies use gate- or RTL-level models, and inject faults into the

entire CPU. In contrast, our study uses a high-level architecture model, and focuses fault injections on the register file, fetch queue, and issue queue only.

Full-CPU fault injection studies demonstrate many faults do not lead to incorrect program execution; instead, they are *masked* and never become visible to software. Several researchers have studied various sources of fault masking. Shivakumar *et al* [1] study masking at the *circuit level*. They develop an electrical and latching-window masking model, and predict the impact these circuit effects have on soft error rates. Kim *et al* [25] study logical masking. They propose "Susceptibility Tables" for logic gates that model the probability a soft error will propagate through a combinational logic block. Mukherjee *et al* [3] identify *microarchitecture-level masking* (mispeculated instructions, predictor structure bits, and microarchitecturally idle bits) as well as *architecture-level masking* (NOP instructions, performance-enhancing instructions, dynamically dead code, and logically masked instructions). Wang *et al* [22] observe certain conditional branch outcomes can be wrong without affecting program correctness, another form of architecture-level masking.

The main difference between our work and all previous studies on soft error susceptibility is the definition of correctness used to judge soft error impact. Previous work requires architectural state to be numerically correct for program execution to be correct; in contrast, our work only requires program outputs to be acceptable to the user. By evaluating correctness at a higher level of abstraction, we measure the additional soft errors that can lead to acceptable program outputs. Like fault masking, our notion of application-level correctness is a form of "fault derating" since it permits certain faults to be tolerable. Instead of derating via fault masking which *hides* faults from the user, we study the faults that are *exposed* to the user, but are derated nonetheless due to the user's willingness to accept some degraded solution quality.

In addition to soft error susceptibility, several researchers have also studied soft computations. Breuer [26, 27] recognizes multimedia workloads can tolerate errors, and proposes exploiting this error resilience to address manufacturing defects. Application-level correctness is similar to Breuer's notion of "error tolerance" (ET) [27], which allows chips that produce numerically incorrect results to be correct as long as their results are acceptable to the user. The main difference is Breuer exploits ET to tolerate hardware defects for higher chip yield, whereas we exploit application-level correctness to tolerate soft errors on functionally correct hardware. Another difference is while Breuer eludes to the importance of capturing "degree of acceptability," we quantify this notion by providing fidelity metrics to directly measure user satisfaction.

Other soft computing research includes Liu *et al* [28] which observes certain image processing and tracking algorithms are inexact, and exploits this to improve task schedulability in real-time systems. Palem [29, 30] exploits probabilistic algorithms to build randomized circuits that are extremely energy efficient. Lastly, Alvarez and Valero [31] exploit the resilience to precision loss exhibited by multimedia applications to develop novel value reuse and energy reduction techniques for floating point operations. Compared to these previous studies, we exploit soft computations for reliability rather than real-time scheduling, energy, or performance.

7 Conclusion

In traditional fault tolerance research, program correctness requires execution to be numerically perfect at the architecture level. However, many programs can appear to execute correctly from the user or application's standpoint, even though execution is not 100% numerically correct. This paper explores definitions of program correctness that view correctness from the application's standpoint rather than the architecture's standpoint. Under *application-level correctness*, a program's execution is deemed correct as long as the result it produces is acceptable to the user. In other words, correctness depends on the user's interpretation of a program's numerical result, not the numerical result itself. To quantify user satisfaction, we rely on application-level fidelity metrics that capture program solution quality as perceived by the user.

We conduct a detailed fault susceptibility study whose goal is to quantify how much more fault resilient programs are at the application level compared to the architecture level. Our conclusion from this study is that a significant number of faults that were previously thought to cause erroneous execution are in fact completely acceptable to the user, especially for programs that produce qualitative results–*e.g.*, soft computations. Across 6 multimedia and AI benchmarks, we find 45.8% of fault injections that lead to architecturally incorrect execution are correct under application-level correctness. For SPEC programs, the increased fault resilience at the application level is lower–only 17.6% of architecturally incorrect faults injected into 3 SPECInt CPU2000 benchmarks produce acceptable results at the application level. This result demonstrates the degree to which programs are more fault resilient under application-level correctness is application dependent; however, we find many programs (including *all* the ones studied in this paper) exhibit some additional fault resilience at the application level.

In addition to studying fault susceptibility, we also present a lightweight fault recovery mech-

anism that exploits the relaxed requirements on numerical integrity provided by application-level correctness to reduce checkpoint cost. After a program crash, our technique only recovers the PC, architected register file, and program stack–we do not restore program text, static data, or heap during recovery rollback–allowing our checkpoints to be very small (only 3 Kbytes on average). Although our lightweight recovery mechanism is not guaranteed to be fail-safe, it successfully recovers 66.3% of program crashes in our multimedia and AI workloads. For SPECInt CPU2000, we can only recover 23.3% to 34.3% of crashes, of which only 2.5% represent additional recoveries allowed by application-level correctness. This lower recovery rate is due to the fewer soft program outputs permitted by SPEC programs compared to soft computations.

8 Acknowledgements

The authors would like to thank Hameed Badawy, Steve Crago, Vida Kianzad, Wanli Liu, Janice McMahon, and Priyanka Rajkhowa for insightful discussions on soft computing. The authors would also like to thank Ming-Yung Ko for help with the SVM-L benchmark. This research was supported in part by NSF CAREER Award #CCR-0093110, and in part by the Defense Advanced Research Projects Agency (DARPA) through the Department of the Interior National Business Center under grant #NBCH104009. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsement, either expressed or implied, of the Defense Advanced Research Projects Agency (DARPA) or the U.S. Government.

References

[1] P.Shivakumar, M.Kistler, S.W.Keckler, D.Burger, and L.Alvisi, "Modeling the effect of technology trends on the soft error rate of combinatorial logic," in *Proceedings of the International Conference on Dependable Systems and Networks*, pp. 389–398, June 2002.

[2] J. Srinivasan, S. V. Adve, P. Bose, and J. A. Rivers, "Lifetime Reliability: Toward an Architectural Solution," *IEEE Micro*, pp. 70–80.

[3] S. S. Mukherjee, C. T. Weaver, J. Emer, S. K. Reinhardt, and T. Austin, "A Systematic Methodology to Compute the Architectural Vulnerability Factor for a High-Performance Microprocessor," in *36th Annual International Symposium on Microarchitecture*, pp. 29–40, December 2003.

[4] M. Gomaa and T. N. Vijaykumar, "Opportunistic Transient-Fault Detection," in *Proceedings of the 32nd Annual International Symposium on Computer Architecture*, pp. 172–183, June 2005.

[5] R. W. Horst, R. L. Harris, and R. L. Jardine, "Multiple instruction issue in the NonStop Cyclone processor," in *Proceedings of the 17th International Symposium on Computer Architecture*, pp. 216–226, May 1990.

[6] N. Oh, P. P. Shirvani, and E. J. McCluskey, "Error detection by duplicated instructions in super-scalar processors," in *IEEE Transactions on Reliability*, pp. 63–75, March 2002.

[7] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. I. August, "SWIFT: Software implemented fault tolerance," in *Proceedings of the 3rd International Symposium on Code Generation and Optimization*, pp. 243–254, March 2005.

[8] G. A. Reis, J. Chang, N. Vachharajani, S. S. Mukherjee, R. Rangan, and D. I. August, "Design and Evaluation of Hybrid Fault-Detection Systems," in *Proceedings of the 32st Annual International Symposium on Computer Architecture*, pp. 148–159, June 2005.

[9] T. N. Vijaykumar, I. Pomeranz, and K. Cheng, "Transient-fault recovery using simultaneous multithreading," in *Proceedings of the 29th annual international symposium on Computer architecture*, pp. 87–98, May 2002.

[10] P. Dubey, "Recognition, Mining and Synthesis Moves Computers to the Era of Tera," *Technology @ Intel Magazine*, pp. 1–10, February 2005.

[11] U. of California at Berkeley, "The Berkeley Initiative in Soft Computing.".

[12] Y. Jin, "A Definition of Soft Computing.".

[13] C. Lee, "MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems," in *Proceedings of the 30nd Annual International Symposium on Microarchitecture*, pp. 330–335, December 1997.

[14] J. Pearl, "Probabilistic reasoning in intelligent systems: networks of plausible inference," Morgan Kaufmann Publishers Inc., 1988.

[15] B. Taskar, M.-F. Wong, P. Abbeel, and D. Koller, "Link Prediction in Relational Data," in *Proceedings of Neural Information Processing Systems*, 2004.

[16] T. Joachims, "Making Large-Scale Support Vector Machine Learning Practical," in *Advances in Kernel Methods: Support Vector Learning*, pp. 169–184, MIT Press, 1999.

[17] V. Kianzad and S. S. Bhattacharyya, "Efficient Techniques for Clustering nd Scheduling onto Embedded Multiprocessors," *IEEE Transactions on Parallel and Distributed Systems*, vol. 17, pp. 667–680, July 2006.

[18] S. Kim and A. K. Somani, "Soft error sensitivity characterization for microprocessor dependability enhancement strategy," in *Proceedings of the 2002 International Conference on Dependable Systems and Networks*, pp. 416–425, September 2002.

[19] N. Wang, J. Quek, T. M. Rafacz, and S. Patel, "Characterizing the effects of transient faults on a high-performance processor pipeline," in *Proceedings of the 2004 International Conference on Dependable Systems and Networks*, pp. 61–72, June 2004.

[20] D. Burger, T. Austin, and S. Bennett, "Evaluating future microprocessors: the simplescalar tool set," in *Technical Report 1308*, University of Wisconsin - Madison Technical Report, July 1996.

[21] C. Chang and C. Lin, "LIBSVM : a library for support vector machines," 2001.

[22] N. Wang, M. Fertig, and S. J. Patel, "Y-branches: When you come to a fork in the road, take it," in *Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques*, pp. 56–67, September 2003.

[23] G. P. Saggese, A. Vetteth, Z. Kalbarczyk, and R. Iyer, "Microprocessor Sensitivity to Failures: Control vs. Execution and Combinational vs. Sequential Logic," in *Proceedings of the 2005 International Conference on Dependable Systems and Networks*, 2005.

[24] E. W. Czeck and D. P. Siewiorek, "Effects of Transient Gate-Level Faults on Program Behavior," in *Proceedings of the 1990 International Symposium on Fault-Tolerant Computing*, pp. 236–243, June 1990.

[25] J. S. Kim, C. Nicopoulos, N. Vijaykrishnan, Y. Xie, and E. Lattanzi, "A Probabilistic Model for Soft-Error Rate Estimation in Combinational Logic," in *Proceedings of the International Workshop on Probabilistic Analysis Techniques for Real-time and Embedded Systems*, (Italy), September 2004.

[26] M. A. Breuer, "Multi-media Applications and Imprecise Computation," in *Proceedings of the 8th Euromicro Conference on Digital System Design*, pp. 2–7, September 2005.

[27] M. A. Breuer, S. K. Gupta, and T. M. Mak, "Defect and Error Tolerance in the Presence of Massive Numbers of Defects," *IEEE Design and Test Magazine*, pp. 216–227, May-June 2004.

[28] J. W. S. Liu, W.-K. Shih, K.-J. Lin, R. Bettati, and J.-Y. Chung, "Imprecise Computations," *Proceedings of the IEEE*, vol. 82, January 1994.

[29] K. V. Palem, "Energy Aware Algorithm Design via Probabilistic Computing: From Algorithms and Models to Moore's Law and Novel (Semiconductor) Devices," in *Proceedings of the International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, pp. 113–116, October 2003.

[30] K. V. Palem, "Energy Aware Computing Through Probabilistic Switching: A Study of Limits," tech. rep., September 2005.

[31] C. Alvarez and M. Valero, "A Fast, Low-Power Floating Point Unit for Multimedia," in *2nd Workshop on Application Specific Processors*, pp. 17–24, January 2003.